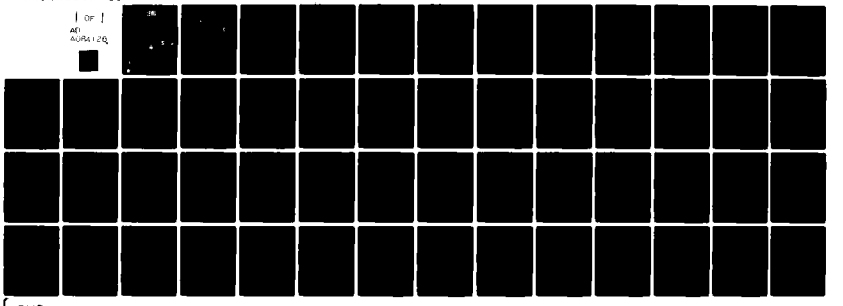
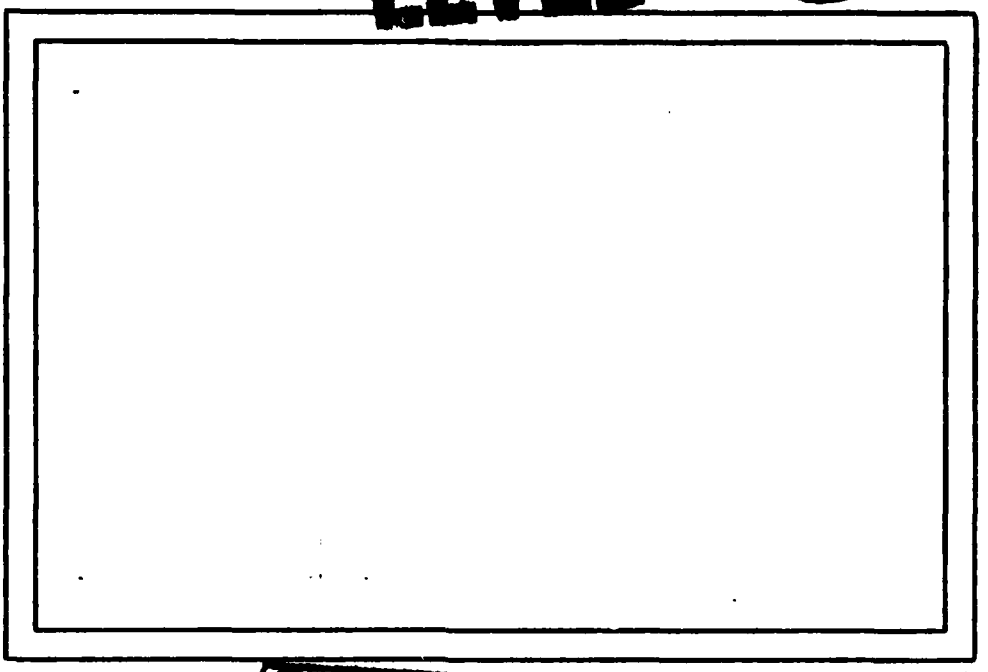
AD-A084 126 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
FUNCTIONS AS DATA OBJECTS: THE IMPLEMENTATION OF FUNCTIONS IN L--ETC(U)
JAN 79 P AGRE N00014-76-C-0477
UNCLASSIFIED TR-726 NL



END
DATE FILMED
6-80
DTIC

LEVEL II



UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

(12) 34

[redacted]
(11) Jan[redacted]79

N00014-76C-0477
NSG-7253

(6) Functions as Data Objects:
The Implementation of Functions in LISP

(10) Philip Agre
Computer Science Department
University of Maryland
College Park, Maryland 20742

(15) N00014-76-C-0477, NSG-7253

Abstract: This paper discusses the manner in which functions are defined and manipulated in various dialects of the *LISP* language. It is argued that the syntax and mechanisms which are usually provided to accomplish such things as passing functions as arguments and returning them as values are unnecessarily inconsistent and clumsy. An alternative scheme for implementing functions in LISP is presented. It is based on the concept of a "linker node", a construct used in Wisconsin/Maryland LISP, and allows greater uniformity in the handling of different types of functions. The advantages of being able to handle functions in the same ways as other common data structures are demonstrated. One possible function syntax is outlined and coded.

(14) TR-726

This work was supported in part by the Office of Naval Research under grant number N00014-76C-0477 and the National Aeronautics and Space Administration under grant number NSG-7253. Their support is gratefully acknowledged.

409022

80 [illegible] 056

Contents

Acknowledgements - Chuck Rieger, Hanan Samet, Milt Grinberg, Steve Small, and Rich Wood read drafts of this paper. The severe errors of reasoning which remain are in spite of their helpful comments. The text was developed using the facilities of the Univ. of Maryland Computer Science Center, and the figures and final text were produced using the facilities of the Univ. of Maryland Computer Science Department.

1. Introduction

The LISP language is a very simple one, having a uniform notation and only a few primitive functions and data structures. In fact, many find LISP so simple that it is often taught in terms of its implementation, complete with drawings of the internal representations of its major data structure, the cons node. One problem that arises from this method of teaching LISP is that it rarely gets to the point where the instructor points out that there are data structures other than cons nodes available in LISP. When reading a LISP program, one often finds great numbers of CAR's, CDR's, and CONS's when the programmer's interpretations are "get the name field", "get the rest of the process id's", and "attach a new frame to this environment".

What is needed in these cases is for the programmer to define an abstract data type such as "personnel record", "process queue", or "environment stack", decide on its representation (that is, its "internal representation") as cons nodes or numbers or whatever, and design a set of functions which allow one to refer to these data types without explicit reference to their representations. Given that it is so easy to do this in LISP, this notion of abstract data types could be a powerful unifying concept in LISP programming.

Our purpose here is to apply the notion of an abstract data structure to LISP functions. This seems natural enough since one

of the major claims for LISP is that programs are represented as data structures which can be built and manipulated under program control. In particular, functions are represented as structures built from cons nodes.

When discussing functions and their implementations, there are several levels of distinction which we would like to make:

1. The abstract mapping between arguments and values,
2. The abstract algorithm which is to carry out this mapping,
3. Our representation of this algorithm as text or a data structure, and
4. The functional object which is actually applied to arguments by an interpreter, that is, our representation of the function as a data structure.

The distinction between the text describing an algorithm (the "external representation") and the functional object derived from this text (the "internal representation") is a useful one because it allows a language designer to describe the syntax of the text in great detail without making any commitments regarding the form which the functional object takes. It is possible for them to take the same form; this is common practice in LISP (provided we identify s-expressions and cons node structures). In a compiled language, the functional object is in assembly language. In any case, leaving the internal representation unspecified allows the designer to select the implementation which is most efficient or elegant or convenient. It is also quite useful when there is more than one possible implementation, such as compiled and interpreted code.

Unfortunately, though, if we consider "function" as naming

an abstract data type, then the problem of failing to distinguish a function from its representation is too often present in implementations of LISP.[1] In general, a function is usually embodied by its LAMBDA expression or by an atomic symbol with which it is identified. This might be explained by noting that, while LISP's notation comes from that of the LAMBDA calculus of Church [Church], the idea of representing functions as cons nodes followed most of the rest of the development of the basis of the LISP language [McCarthy2]. The notion that "functions are different from data" carries down to the present day, despite the semantics of the LAMBDA calculus.[2]

The thesis of this paper is that there should be developed a notation and implementation of functions which treats them in just the same manner as cons nodes or numbers, to the extent that it makes sense for functions to have the properties of cons nodes and numbers. This has several consequences. The first is that one should be able to pass a function as an argument, return it as a value, or assign it to a variable in the same manner (and using the same mechanisms) as other data objects. The second is that a function should be a data object independent of its external representation (as, say, a lambda-expression), just as numbers are not stored in the same manner as they are printed.

---

[1]The case of MacLISP is discussed in detail in Section 2.

[2]Several other features of LISP do not agree with the semantics of the lambda calculus. See [Steele] for a further discussion of this point and an interesting application of lambda calculus techniques to compiler optimization.

It is true that functions do not have all the same properties as our exemplary data objects, cons nodes and numbers. For example, one cannot easily test two functions for equality. However, this is no more an essential property of a data structure than is the existence of a natural order. This latter property is true of numbers but not of cons nodes, but does not in any way demean cons nodes' claim to being legitimate data structures.

Section 4 will present an implementation of functions which has all these properties. Section 5 suggests a syntax for function definition and manipulation and shows how it would be implemented using the notions of Section 4. Later sections proceed to explore other consequences of these ideas; in particular how programming styles might be affected by the introduction of a "theoretically sound" implementation of functions. Other issues relating to the implementation of functions are discussed; in particular, a scheme for organizing the association list is presented which ends the need for specifying different function types for functions which can handle optional arguments (e.g., the LIST function).[3]

---

[3]To avoid appearing unduly pedantic, phrases such as "the X function" will be understood to mean "the function bound to X", especially when one usually identifies that function with X, as is the case with LIST, SETQ, LAMBDA, etc. This is in spite of the fact that this is a crucial distinction here.

2. Traditional Implementations of LISP Functions

Most of the major implementations of the LISP language differ with the views expressed above with respect to the definition and manipulation of functions. We shall consider the MacLISP implemenation of LISP in detail here. MacLISP was a direct descendent of the first important implementation of LISP, and most of the other major implementations in use today share its basic mechanisms for defining and manipulating functions.

In MacLISP, there are two ways to create a function. The first is to use DEFUN or more primitive means to attach a function definition to an atomic symbol by placing it on the symbol's property list under one of the indicators: EXPR, FEXPR, or MACRO. The representation which is placed under these indicators is a LAMBDA expression which was created by DEFUN or a user-defined routine. The interpreter, given an atomic symbol as the CAR of an s-expression to be evaluated, looks down the symbol's property list for the first occurrence of one of these properties. If one is found then the associated LAMBDA expression is used. Dynamically bound functions ("funargs") created by FUNCTION or *FUNCTION are bound to atoms as their values.[4] These funargs have the form (FUNARG <function-rep>

---

[4]For convenience, we will treat "x is bound to y" as a symmetric relation. Anyone having a preference as to which is the proper order of arguments to "is bound to" may translate.

{. <BCP>}), where BCP is a representation of an environment known as a Binding Context Pointer, essentially an offset into the stack which allows the evaluator to reconstruct the saved environment in the variables' shallow-binding "value cells". Because of MacLISP's stack-like environment mechanism, it is not possible to return a FUNARG with a BCP outside of the dynamic scope of the environment in which it was created. The FUNCTION function is essentially the same as the QUOTE function. Indeed, they are implemented in the same manner. The only need for a distinction is so that the MacLISP compiler can have the information that the argument is indeed a function.

MacLISP functions have several interesting properties. First, given a data object, it is not in general possible to determine whether it is a function or whether it was intended for some other purpose. This is not as bad as it sounds; any composite data structure which is represented as a group of cons nodes will have the same difficulty. There should be no problems if the programmer has made an assignment of abstract data types to all of the program's variables and function arguments.

Second, and more significantly, we see several cases in which different types of functions are unnecessarily handled through different mechanisms. For example, a FEXPR's arguments are passed to it as a list which is bound to its first specified argument name. If a second argument name is provided, it is bound to the environment in which the function is being invoked in case the function should want to evaluate one of the other

arguments in an environment free of its local variables. For example,

```
(DEFUN SETQ FEXPR (ARGS ENV)
     (SET (CAR ARGS) (EVAL (CADR ARGS) ENV)))
```

Also, it happens that only the lowliest of functions (the ordinary EXPR, possibly with a *FUNCTION environment), can exist independently of an atomic symbol.

Third, and most importantly for our purposes, we see that there are several ways in which a data object can be bound to an atomic symbol; indeed two different objects can be bound to a symbol at one time in the same environment. The only use which MacLISP makes of this capability is a trick which has a symbol bound function-wise to a function and value-wise to some binary indication of whether the function (or the function package which it represents) is loaded and its feature enabled. This is little justification for so completely separating the handling of functions from the handling of other data objects, especially considering the fact that function binding using property lists is not sufficient for all purposes.[5]

---

[5]For an interesting insight into the origins of property-list definition of functions, compare the discussion of Section 8 to [McCarthy1], Section 2.3.

3. A Notion of "Function as Data Object"

If the conditions regarding the implementation of functions which were stated in the Introduction are to be satisfied, a kind of data structure must be designed to embody individual functions without regard for the entity to which the function is bound or how it was first created. This structure must include all the information which is necessary to apply the function to a set of arguments.

In LISP, there are several kinds of properties a function can have. The two most obvious are whether or not its arguments should be evaluated and how its body is to be evaluated (e.g., the bodies of expr's and fexpr's are evaluated once, whereas those of macro's are evaluated, and the resulting expression is also evaluated). Other properties include an environment assigning values to some or all of the function's free variables, a "label" for unassigned ("anonymous") functions, and perhaps a trace routine to print messages on entry and exit. The usual way of distinguishing the subtleties of functions' definitions (i.e., by assigning properties to the atom with which the function is associated) is in obvious conflict with our assumptions about the nature of functions. One approach to the problem of defining a function's many properties would be to declare them all at definition time, possibly by adding extra entries in the LAMBDA form. This is not sufficient, however, since we often wish to add a property to a function after it has been defined, or to

have two versions of a function with different properties. This and the desire of the LISP maintainer to add new function properties conveniently seem to preclude the possibility of representing a function as a data node with several "slots", one for each property.

Many problems of functions as data objects such as attachment of properties to functional objects can be solved by appropriating the notion of a "linker node" from Wisconsin LISP [Norman]. The basic structure of a linker node is depicted in Figure 1. The first word of the node is an instruction of the machine language (denoted "SCALL <addr>" here) which loads the address of "Word 1" into some index register and jumps to the address indicated in the instruction's address field.[6] For a given function type, the routine at this address is called the type's "expansion routine". The information pointed at by the two halves of word 1 may include a LAMBDA-expression, an association list (or other data structure representing an environment), or another function node, depending on the function type.

Linker nodes are created by "typing functions" which are provided by LISP and available to the user. Two familiar typing functions are FUNCTION and LABEL. A typing function takes as its argument a function (or in some cases an argument list and s-expression) and constructs a linker node which points at that type's expansion routine and the various information which will later be needed by the expansion routine.

---

[6]This is the standard subroutine-call instruction in non-stack architectures; most systems can adapt this structure easily using whatever subroutine-call instruction is available.

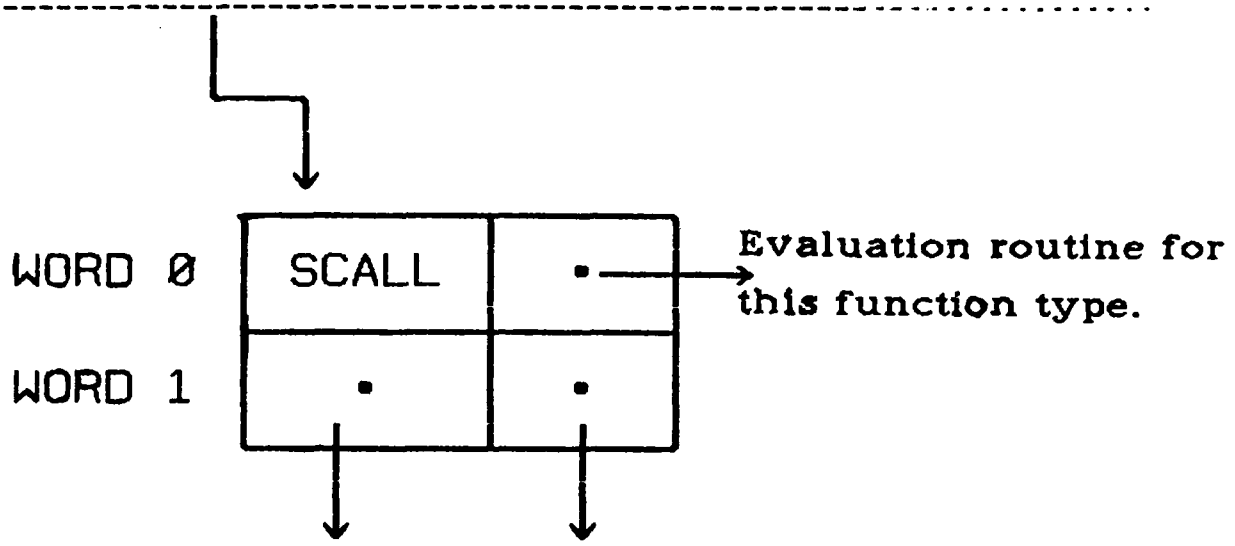


Figure 1

A complicated function may consist of a long chain of linker nodes, each of which describes the setting up of one part of the environment in which the function is to be executed. In the usual case, the code to be executed by jumping to a linker node should be just that which would have been required by an interpreter of the traditional design, say, evaluate the arguments and bind them on the alist, then evaluate each expression in the function's body, returning the value of the last one. The only difference between the traditional and linker node schemes in this case would be the locations of the various routines to perform these actions. Since each linker node contains the address of the routine it calls, determining what type of function is being called does not contribute any extra overhead. Thus the linker node function implementation can be

more efficient than traditional schemes, though that is not the major argument in its favor.

4. A Possible Manifestation of the Scheme

Let us propose a set of typing functions and show how they would be implemented through their expansion routines and the evaluator. By a "regular form", we mean a function whose arguments are to be evaluated and whose body is to be executed in the usual way. A "special form" is defined to be the same as a regular form, except that its arguments are not to be evaluated by the interpreter. These are the EXPR's and FEXPR's of MacLISP. Given functions f and g (which may be either abstract or concrete), let f $\circ$ g denote the function which is their composition, so that (f$\circ$g)(x)=f(g(x)). Finally, let "<var> <-- <val>" denote "IF <var>=Ø THEN <var> <- <val>".

(LAMBDA <argnames> <sexp1> ... <sexpn>) creates a regular form with the given arguments and body. A more precise discussion of LAMBDA's syntax is given in Section 9. The expansion routine is EXPAND.

(EXPR <fn>) takes a function <fn> and returns a version of it which is a regular form. The expansion routine is EEXPAND.

(FEXPR <fn>) takes a function <fn> and returns a version of it which is a special form (aka fexpr). The expansion routine is FEXPAND.

(MACRO <fn>) takes a function <fn> and returns a version of it which is a macro. The expansion routine is MEXPAND.

(FUNCTION <fn>) takes a function <fn> and returns a version of it which has the current environment "trapped" on it, in the traditional "funarg" manner. The expansion routine is ENVEXPAND.

(LABEL <at> <fn>) takes an unevaluated atomic symbol <at> and an evaluated function <fn> and returns a function g such that whenever g is called, <at> has the binding <fn>. The expansion routine is LEXPAND.

All of these are regular LISP functions which return functional objects (i.e., linker nodes) as values. In particular, LAMBDA need not be handled specially by the interpreter; it is a special form of two or more arguments which returns a function as its value.

These functions should have the following properties:

(i) For t and u in {EXPR, FEXPR, MACRO},
t o u has the same effect as t
t o FUNCTION has the same effect as FUNCTION o t

(ii) If f is the result of several applications of FUNCTION, the least recently (first) trapped environment is the one to be used when the function's body is evaluated.

(iii) LABEL assignment should hold regardless of whether the function is the result of FUNCTION. At most one LABEL may be assigned.

(iv) Any FUNCTION or LABEL environment bindings should not take effect until after any evaluation of arguments has taken place.

The pseudo-code to be presented below shows how these functions and their expansion routines might be implemented on a standard machine. For the purpose of these illustrations, let us make some defining assumptions about the LISP of which it is supposedly a part:

1. The binding mechanism used is deep binding.

2. The alist is implemented as a list of elements of the form

(<ats> . <vals>), where <ats> is a list of atomic symbols and <vals> is a list of the atomic symbols' values. This implementation of the alist mechanism is interesting in itself, and Section 9 is dedicated to describing it.

3. System-defined functions such as CAR and CONS are represented as linker nodes, except that there is a pointer to machine language code instead of to an s-expression to evaluate.

4. The function evaluator pushes the current environment on the system stack and the function exit routine pops it.

None of these assumptions is particularly critical in a linker-node scheme. A shallow-binding system is bound to be more complex, and linker nodes have little to do with simplifying (or complexifying) either binding mode, though the modularity they introduce will make any scheme somewhat easier to modify.

Let us assume the presence of several global variables:

args originally contains the list of expressions which were given as arguments to the function being processed. When it is decided what should be done with them (i.e., whether they should be evaluated), they are moved to evargs.
evargs is set to the list of argument values, once they have been determined. At all times, exactly one of {args, evargs} is non-zero.
envptr points at the current alist.
altenv is used to hold the funarg environment for the function being evaluated.
labpair is used to hold the binding specified by LABEL, i.e., a dotted pair (<name> . <function>).
contptr is used to hold the address of the routine which is to receive control once the s-expressions to evaluate and the environment to use are found.
value is used to hold the s-expressions to be evaluated while control is being transferred to (contptr) and to return a function's value.

Given these assumptions, the typing functions and expansion routines named above could be implemented as follows:

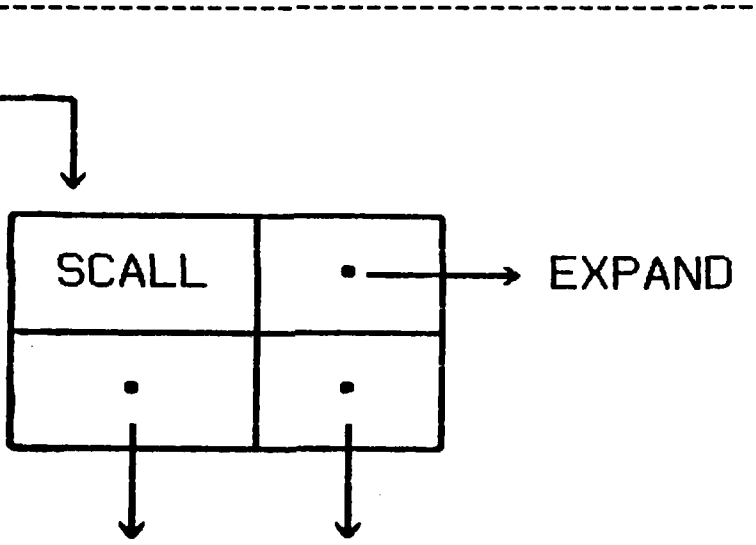


Figure 2

(LAMBDA <argnames> <sexp1> ... <sexpn>) creates the linker node in Figure 2, where EXPAND is:

```
IF args ≠ 0
  THEN PUSH contptr, altenv, and labpair[7]
       evaluate the members of args
       evargs <- a list of these values
       POP contptr, altenv, and labpair
ENDIF
envptr <- cons(cons(<argnames>,evargs),
                  IF altenv ≠ 0 THEN altenv ELSE envptr)
IF labpair ≠ 0
```

[7]This and the simulation program in the appendix suggest that these variables should be assigned fixed offsets from the current stack top so that each EVAL invocation will have its own distinct set of pointers. If the stack is one word wide, then these variables will probably occupy two words per invocation on the stack.

```
  THEN envptr <- cons(labpair,envptr)
ENDIF
value <- (<sexpl> ... <sexpn>)
contptr <-- EVALIT
JUMP TO (contptr)

EVALIT:
IF value points at compiled code
  THEN JUMP TO (value)
  ELSE args <- value
       JUMP TO DO
ENDIF
```

DO is a function which expects a list of forms to arrive in the args variable and which evaluates each, returning the value of the last one.

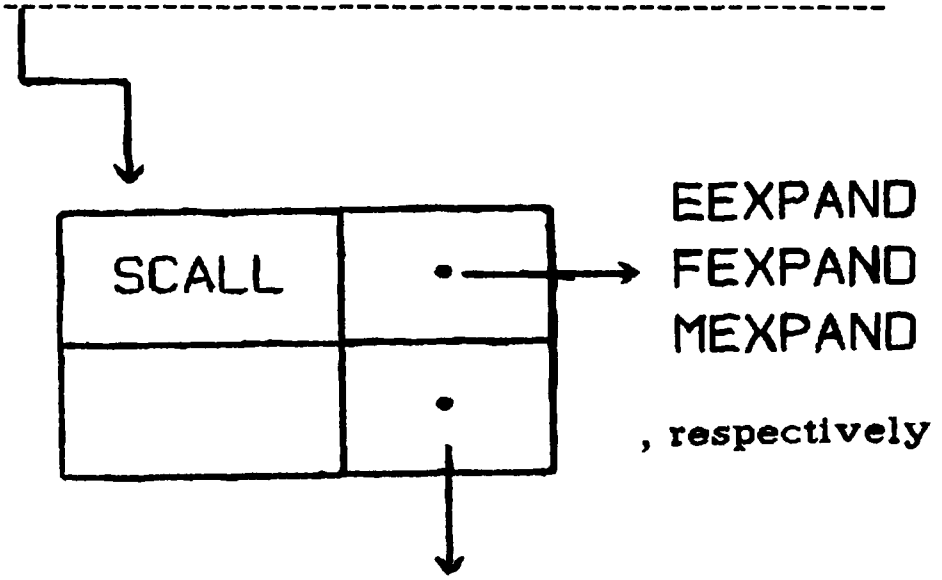


Figure 3 <fn>

(EXPR <fn>), (FEXPR <fn>), and (MACRO <fn>) create the structure in Figure 3, where EEXPAND is:

```
IF args ≠ Ø
  THEN PUSH altenv, label, and labval
       evaluate each entry on args
       evargs <- a list of these values
       args <- Ø
       POP altenv, label, and labval
ENDIF
contptr <-- EVALIT
JUMP TO <fn>
```

FEXPAND is:

```
IF args ≠ Ø
  THEN evargs <- args
       args <- Ø
ENDIF
contptr <-- EVALIT
JUMP TO <fn>
```

and MEXPAND is:

```
IF args ≠ Ø
  THEN evargs <- args
       args <- Ø
ENDIF
contptr <-- MACEVAL
JUMP TO <fn>

MACEVAL:
IF value points at compiled code
  THEN call the routine at value
  ELSE call DO(value)
ENDIF
args <- value
JUMP TO EVAL
```

EVALIT and MACEVAL are the continuation routines for their respective function types. The reader is urged to make a careful comparison between the EXPAND, EEXPAND, FEXPAND, and MEXPAND routines. We note that EXPR's and FEXPR's differ in the way they handle their arguments but not in the way they evaluate their expressions, whereas FEXPR's and MACRO's differ not in the way they handle their arguments, but in the way they evaluate their expressions. This is reflected in the ways in which their

expansion routines manipulate the internal variables. For example, both the FEXPAND and MEXPAND routines have code to move the arguments from args to evargs unevaluated if they have not already been processed, but the EEXPAND routine specifies that the arguments be evaluated before being moved to evargs. Also, note that the EEXPAND and FEXPAND routines reference the same continuation pointer, EVALIT, while MEXPAND references a different one, MACEVAL. What is more, with a little ingenuity, the EEXPAND routine could be made part of the EXPAND routine. Also, note that the three latter expansion routines have nothing to do with the nature of the environment in which the function is executed; this is the domain of the expansion routines of functions like FUNCTION and LABEL.

(FUNCTION <fn>) builds a linker node whose expansion routine is ENVEXPAND and which points at the current environment and at <f>. The ENVEXPAND routine is:

```
altenv <- <environment>
JUMP TO <fn>
```

Note that the since the innermost FUNCTION environment is the one to be used, ENVEXPAND does not check whether altenv=Ø before assigning to it.

(LABEL <at> <fn>) builds and returns a linker node whose expansion routine is LEXPAND and which points at both <at> and <fn>. The LEXPAND routine is:

```
labpair <- cons(<at>,<fn>)
JUMP TO <fn>
```

We also define FLAMBDA to have the same effect as FEXPR o LAMBDA, MLAMBDA to have the same effect as MACRO o LAMBDA, and F/L to have the same effect as FUNCTION o LAMBDA. (The writing of these functions and their expansion routines, FEXPAND1, MEXPAND1, and ENVEXPAND1, is straightforward, and not included here.)

Having defined the expansion routines, the evaluator only has to handle atoms, initialize the global variables, and jump to the function to which the CAR of the form evaluates.

```
EVAL(sexp)
IF sexp is an atomic symbol
  THEN return(lookup(sexp,envptr))
  ELSE IF sexp is not a cons node
         THEN return(sexp)
         ELSE PUSH envptr     /* save environment for posterity */
              contptr, altenv, and labpair <- 0
              push(cdr(sexp))
              set fn <- EVAL(car(sexp))
              pop(args)
              JUMP TO fn
       ENDIF
ENDIF
```

Some of the traditional jobs of EVAL, such as the evaluation of arguments, have been moved elsewhere in the interpreter. Others, such as checking for LAMBDA forms and determining what type of function is being called, have been eliminated by the structure of the functions' representations.

5. Some Examples and Consequences

We have now written most of a LISP interpreter using the "function as data object" approach outlined in the introduction. Let us consider some examples which demonstrate the power and flexibility of this approach.

Given SET, a regular form of two arguments such that (SET <var> <val>) assigns <val> as the value for <var>, we can define the traditional value and function definition mechanisms:

```
(SET 'SETQ (FLAMBDA (NAME SEXPR)
     (SET NAME (EVAL SEXPR))))

(SETQ DEFUN (FLAMBDA (NAME TYPE ARGS SEXPR)
     (SET NAME ((EXPR (COND ((EQ TYPE 'EXPR) LAMBDA)
                            ((EQ TYPE 'FEXPR) FLAMBDA)
                            (T MLAMBDA)))
                ARGS SEXPR))))
```

The fully indoctrinated user of MacLISP or InterLISP might recoil at the latter definition. Its comprehension admittedly requires more than one reading. Let us draw some pictures to explain what goes on in this DEFUN. The value of the atomic symbol DEFUN is depicted in Figure 4. Suppose we call (DEFUN FOO FEXPR (X Y) (CONS Y X)). Then the FEXPAND routine, seeing args=(FOO FEXPR (X Y) (CONS Y X)), sets evargs <- (FOO FEXPR (X Y) (CONS Y X)) and args <- Ø, so that EXPAND will not evaluate the arguments. EXPAND, not having any arguments to evaluate, creates the new association list segment depicted in Figure 5. EXPAND then sends the body of DEFUN to the evaluator. In the course of the

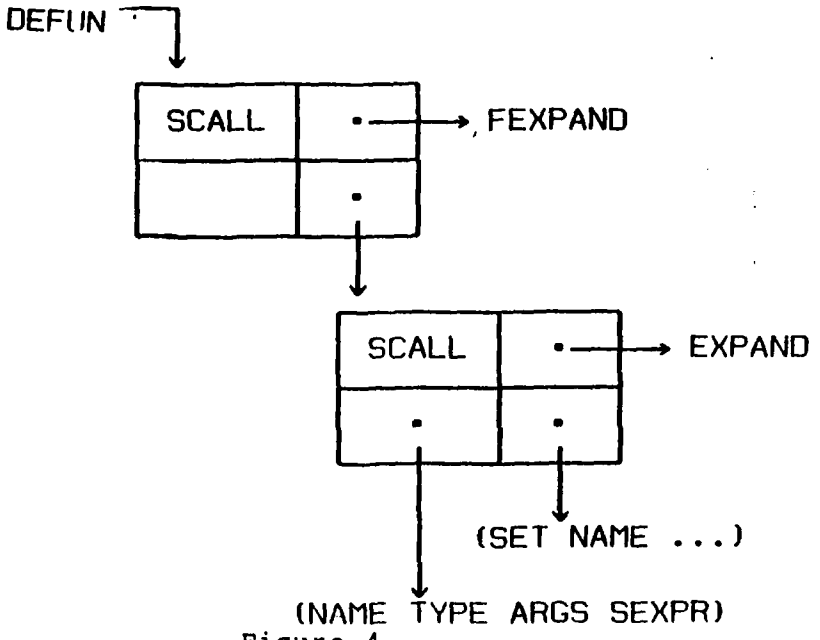


Figure 4

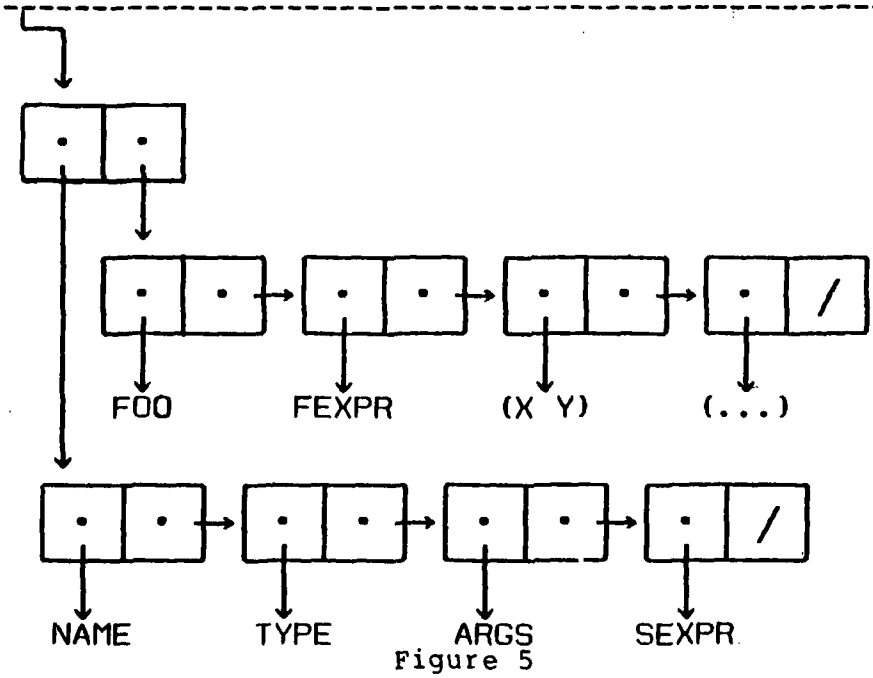


Figure 5

evaluation, COND, upon finding that TYPE is bound to FEXPR,

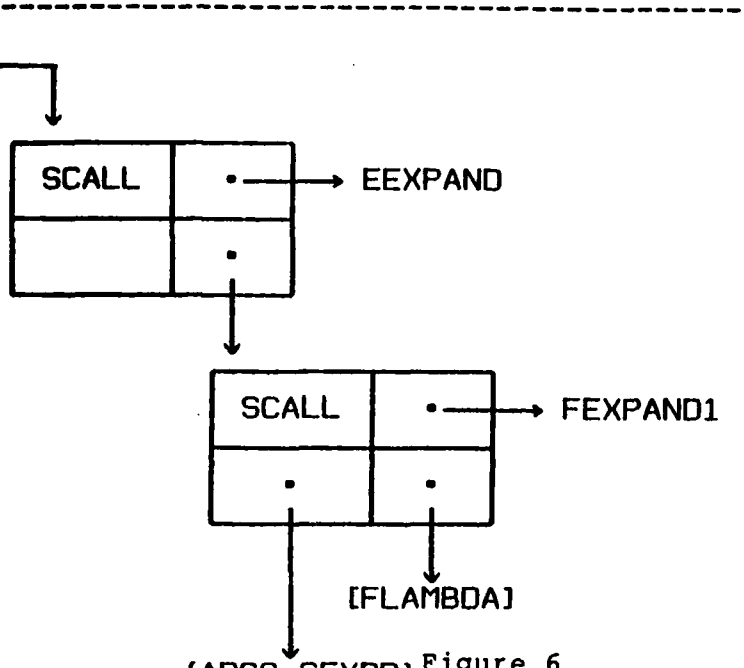


Figure 6

returns the FLAMBDA function (ie, the value of the atomic symbol FLAMBDA) as its value. This function (consisting of one linker node which points at FEXPAND1) is fed to EXPR to obtain the structure in Figure 6 (in which [<at>] stands for the machine language code which executes the function associated with the atomic symbol <at>).

Now, args is assigned the value (ARGS SEXPR) by EVAL. These are evaluated by EEXPAND to obtain ((X Y) (CONS Y X)) in evargs, with args being set to Ø. FEXPAND1, finding args=Ø, assumes that evargs contains the properly handled arguments and creates the new segment of the association list depicted in Figure 7. The code for FLAMBDA proceeds to create and return the structure depicted in Figure 8. To conclude the evaluation, SET assigns this to FOO, which is the value of NAME.

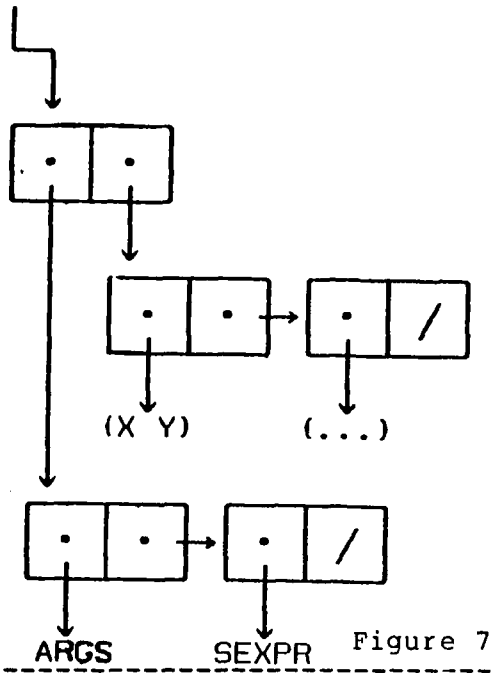


Figure 7

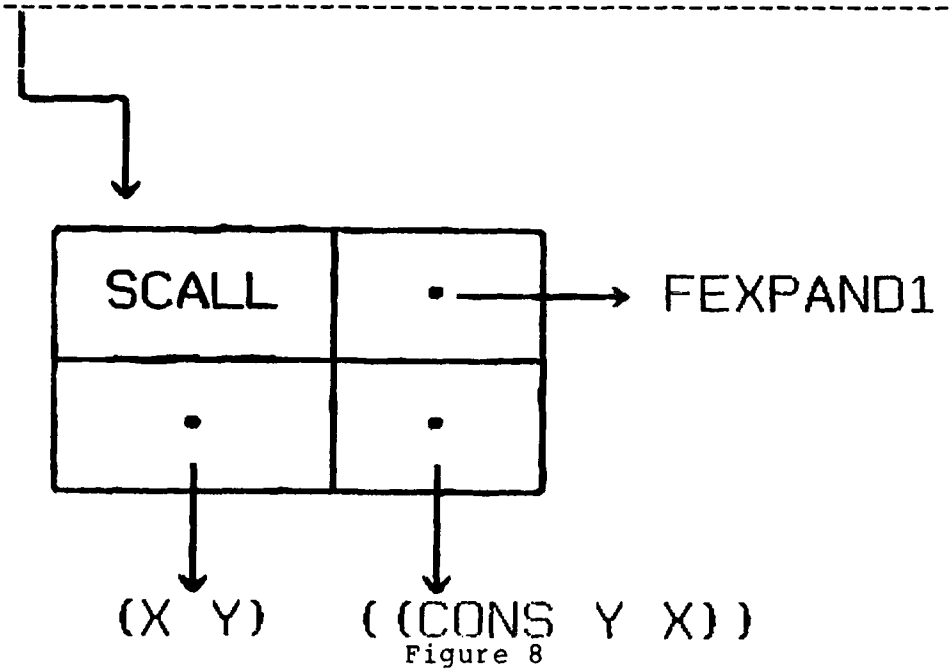


Figure 8

Similar techniques can be developed for such definitions as:

```
(SETQ XYZ (LABEL MAPC (LAMBDA (L FN)
                (COND ((ATOM L) NIL)
                      (T (CONS (FN (CAR L))
                               (MAPC (CDR L) FN)))))))
```

with the call:

```
(XYZ '((A B) (C D)) (F/L (L) (XYZ L PRINT)))
```

6. Addition of Function Types

One of the primary advantages of this scheme is seen only by the LISP maintainer, who is periodically asked to implement new kinds of evaluation schemes. Two familiar examples, CLOSURE and TRACE, are sketched here.

(CLOSURE <var-list> <fn>) returns a version of the function <fn> which behaves similarly to (FUNCTION <fn>) except that only the values of the variables in <var-list> are trapped; the rest remain dynamically scoped.

In one implementation, (CLOSURE <var-list> <fn>) could create and return the structure depicted in Figure 9, where <val-list> is a list of the values of the variables in <var-list>, and CEXPAND is:

```
IF cllist ≠ 0
  THEN cllist <- cons(cons(<var-list>,<val-list>),cllist)
  ELSE cllist <- list(cons(<var-list>,<val-list>))
ENDIF
JUMP TO <fn>
```

The interpreter is easily extended to allow CLOSUREs; all we need to do to the code presented above is:

(i) Put "cllist <- 0" with the other initializations in EVAL,

(ii) NCONC the alist segments on cllist onto the front of the association list to be used when executing the function, and

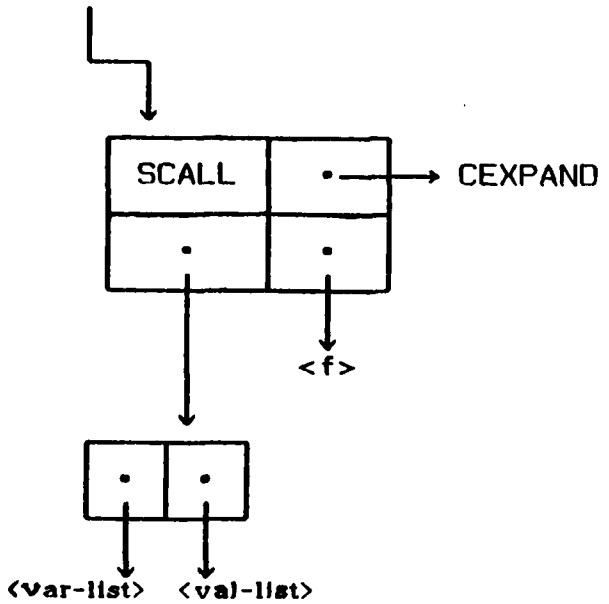


Figure 9

(iii) Be sure to push and pop cllist when evaluating arguments to functions in expansion routines like EXPAND and EEXPAND.

Note that, given this implementation, LABEL can be implemented using the same mechanism as CLOSURE if LEXPAND is:

```
IF cllist ≠ Ø
  THEN cllist <- cons(cons(<atm>,<fn>),cllist)
  ELSE cllist <- list(cons(<atm>,<fn>))
ENDIF
```

(TRACE <f> <g>) is a generalized function-tracing function. If one wishes some action to be performed on every entry and exit to <f>, then (TRACE <f> <g>) will return a version of <f> for which this action is to be performed by <g>. For example, <g> could print "ENTERING F" and "LEAVING F". The function <g>

should be prepared to receive as arguments the function <f> and a list of the arguments which were given to the traced version of <f>.[8] The value of (TRACE <f> <g>) is the structure depicted in Figure 10, where TREXPAND is:

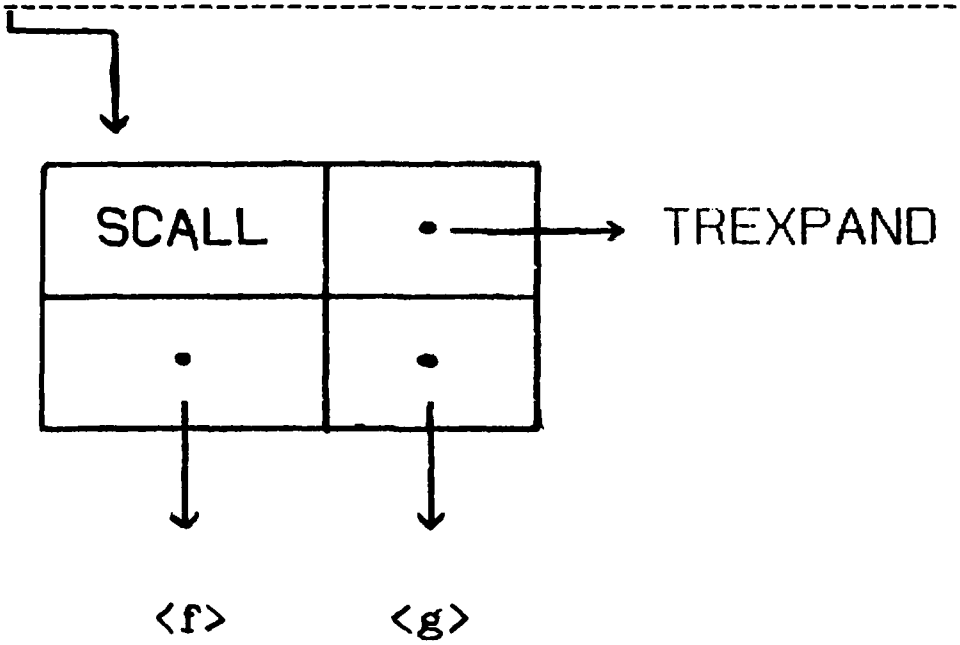


Figure 10

evargs <- list(<f>,IF args=0 THEN evargs ELSE args)
JUMP TO <g>

The function <g> could print a message and then call <f> with the given argument list. To let <g> know <f>'s "identity", we could do:

```
(SETQ TRACER (LAMBDA (NAME FN)
     (TRACE FN (F/L (FN ARGS)
          (PRINT "CALLING " NAME " WITH ARGS:" ARGS)
          (APPLY FN ARGS)))))
```

The writing of APPLY is left as an easy exercise. Note that this APPLY is substantially less complicated than more traditional

[8]This design for a function tracing mechanism is part of the Wisconsin LISP system.

APPLY's. However, it must be decided for each implementation what it means to give a special form or macro to APPLY.

7. Computing with Functions

Now that we have designed an efficient, general facility for handling functions of all types, one might ask whether there are any benefits to be derived. What is presented here is one way in which one's programming style might change given such a facility.

A common sort of function in LISP takes a function and an s-expression (usually a list) as arguments and applies the function several times to different parts of the s-expression. For example, the MAPCAR function takes as its arguments a function and a list and applies the function to each element of the list, returning a list of the results. The MAPCAR function might be written as:

```
(SETQ MAPCAR (LAMBDA (FN LST)
     (COND ((NULL LST) NIL)
           (T (CONS (FN (CAR LST)) (MAPCAR FN (CDR LST)))) )))
```

There is another way of looking at this. Suppose we had a function VECTORIZE such that (VECTORIZE <fn>) returned a function which would take a number of arguments and apply <fn> to each one, returning a list of the values obtained. Then, instead of writing (MAPCAR LIST '(A B C)), one could write ((VECTORIZE LIST) 'A 'B 'C), and obtain the value ((A) (B) (C)). Given the scheme described above, VECTORIZE is easily written[9] as:

---

[9]The CLOSURE's used in these functions are not necessary except for efficiency and clarity of exposition. To apply one of

```
(SETQ VECTORIZE (LAMBDA (FN)
     (CLOSURE '(FN) (LAMBDA ARGS (MAPCAR FN ARGS)))))
```

Similarly, consider the INDEX function, which takes as its arguments a function f of two arguments and a list l = (x1 ... xn) which contains at least two entries and computes f(x1,f(x2, . . ., f(xn-2,f(xn-1,xn)) . . .)). If f has reasonable properties (all that is really required is that its result be of the same data type as its arguments), then this essentially forms a generalization of f to any legal number of arguments. The INDEX function can be written as:

```
(SETQ INDEX (LAMBDA (FN LST)
     (COND ((NULL (CDR LST)) (CAR LST))
           (T (FN (CAR LST) (INDEX FN (CDR LST)))) )))
```

If we think of INDEX as performing some action on f relative to the list l, then what we would really like is a function INDEXIFY, which would take f as an argument and return a version of f which can take any number of arguments. This is easily written as:

```
(SETQ INDEXIFY (LAMBDA (FN)
     (CLOSURE '(FN) (LAMBDA ARGS (INDEX FN ARGS)))))
```

This sort of definition has considerable potential for improving the readability of code. Assuming that the reader trusts in INDEXIFY's ability to efficiently generalize functions in the right way, one need only write the 2-argument version of any

---

these operators to a function with free variables, it should probably be FUNCTION-ized. Also note that the LAMBDA syntax described in Section 9 is being used.

desired function and then apply INDEXIFY to it. For example,

```
(SETQ MIN (INDEXIFY (LAMBDA (X Y)
     (COND ((LESSP X Y) X) (T Y)))))
```

is sufficient to define an n-argument MIN function.

A common and useful operation in Mathematics is to generalize the definition of a function on numbers to a function on functions on numbers (and occasionally to a function on functions on functions on numbers). For example, given functions f and g on R, it is common to define f+g by (f+g)(x)=f(x)+g(x) for all x or max(f,g) by max(f,g)(x)=max(f(x),g(x)) for all x. We can define a function FGENERALIZE to implement this notion of generalization as follows:

```
(SETQ FGENERALIZE (LAMBDA (FN)
     (CLOSURE '(FN) (LAMBDA FUNCS
          (CLOSURE '(FN FUNCS) (LAMBDA ARGS
               (APPLY FN (MAPCAR FUNCS (LAMBDA (FUNC)
                    (APPLY FUNC ARGS)))))))))
```

Given this definition, we can make these definitions:

```
(SETQ FLIST (FGENERALIZE LIST))
(SETQ ARITHOPS (FLIST PLUS TIMES DIFF QUOTIENT REM))
```

and compute: (ARITHOPS 9 4) = (13 36 5 2 1).

To complete our set of functional operators, we can define COMPOSE so that (COMPOSE f g) = f $\underline{o}$ g and STACKIFY so that ((STACKIFY f) '(x1 ... xn)) = (f x1 ... xn) by:

```
(SETQ COMPOSE (LAMBDA (FN1 FN2)
     (CLOSURE '(FN1 FN2) (LAMBDA ARGS
          (FN1 (APPLY FN2 ARGS))))))
```

```
(SETQ STACKIFY (LAMBDA (FN)
     (CLOSURE '(FN) (LAMBDA (LST) (APPLY FN LST)))))
```

These operators provide us with an alternate way of defining functions that is in many ways more appealing than the traditional method of writing calls on MAPCAR and APPLY in many places in the code. As an example of these operators, to define the function "sin squared plus cos squared", we write:

```
(SETQ SIN2COS2 ((FGENERALIZE PLUS)
                    (COMPOSE (LAMBDA (X) (TIMES X X)) SIN)
                    (COMPOSE (LAMBDA (X) (TIMES X X)) COS)))
```

or

```
(SETQ SQUAREFN (LAMBDA (FN)
     (COMPOSE (LAMBDA (X) (TIMES X X)) FN)))
(SETQ SIN2COS2 ((STACKIFY (FGENERALIZE PLUS))
                  ((VECTORIZE SQUAREFN) SIN COS)))
```

In contexts where we think of ourselves as modifying or operating on functions, this type of programming can ease both the reading and writing of programs. However, a programmer will only use a facility such as this if it is known to be both convenient and efficient. Allowing a programmer this flexibility is one of the major reasons for adopting an implementation of functions (or of anything, for that matter) which is uniform in its notation and execution.

8. What is Meant by "Referring to a Function"?

A problem which is commonly encountered by LISP programmers is exemplified by this conversation:

```
> (SETQ FACT (LAMBDA (N)
       (COND ((LESSP N 2) 1)
             (T (TIMES N (FACT (SUB1 N)))) )))
>> "DONE"
"FACT is bound to the factorial function."
> (FACT 4)
>> 24
"Good, it works."
> (SETQ FACTORIAL FACT)
>> "DONE"
"Now FACTORIAL is the factorial function."
> (FACTORIAL 5)
>> 120
"Good, it works."
...
> (SETQ FACT (LAMBDA (TEMPLATE)
       (LOOKUP TEMPLATE MAIN-DATA-LIST)))
> "DONE"
"Now FACT is the main data-base lookup function."
>> (FACT '(LOVES ?X ?Y))
> ((FACT-27 (X . JOHN) (Y . MARY))
   (FACT-92 (X . MARY) (Y . BILL)))
"Good, it works."
>> (FACTORIAL 3)
> "CAN'T MULTIPLY BY NIL"
"Oops, what happened?"
```

What happened, of course, is that the LAMBDA-expression which is bound to FACTORIAL makes reference to FACT rather than to FACTORIAL when recursing. The functional object which was originally bound to FACT is not really the factorial function in the manner in which we would like to consider it. It is only the factorial function when it is bound to FACT; when something else is bound to FACT, it becomes something different.

It would appear that, in this case at least, we have not completely succeeded in separating internal and external representations of functions (i.e., it is necessary for FACT to represent the factorial function). There exists a construct which will patch this example. It is called LABEL, and was once widely used to write unbound ("anonymous") LAMBDA-expressions which were recursive. The implementation of LABEL in the linker node scheme is straightforward and is described in Section 4. What is needed in the FACT/FACTORIAL example above is to make the fact that FACT is the factorial function part of the definition of the factorial function itself. We can use the LABEL construct to achieve this end by writing:

```
(SETQ FACT (LABEL FACT (LAMBDA (N)
     (COND ((LESSP N 2) 1)
           (T (TIMES N (FACT (SUB1 N))))))))
```

Having done this, we can give FACT's binding to FACTORIAL and rebind FACT, and FACTORIAL will still be bound to the factorial function. This is because the FACT reference in the factorial function refers to the LABEL FACT, not to the SETQ FACT.

While this works in this one case, the whole world is not as simple as the factorial algorithm. The typical program is written as a large collection of function definitions (i.e., bindings of function objects to atomic symbols) which refer to one another in complex patterns. For example, it often happens that two functions will refer to one another; they are said to be "mutually recursive". The usual structure for a large program is for there to be a large number of small "primitive" operations,

usually performing the basic operations on various kinds of data structures or summarizing frequently performed actions. Most of the rest of the functions use these primitive ones. The rest of the functions are usually grouped roughly hierarchically, with a few "top level" functions calling their "helper functions" which call their helper functions and so on. The top level functions form the interface to the outside world. If the program is being presented as a "package" for other users, then all that is described is the set of top level functions which are to be referenced by calling programs as their primitives. As far as the calling program is concerned, these functions perform some (probably quite complex) operations which are defined in terms of their effects rather than their algorithms. In a sense, all the lower members of the hierarchy are clauses of the top level functions; they have no meaning except as referenced by their given names by their initially defined callers. Such collections of definitions are prone to having their members' names accidentally re-defined by another programmer who is using the package. One usually attempts to prevent this by encoding the names of the functions (in the appendix, many variable and function names have X prefixes in the hope that calling functions' names won't start with X). This is not quite a satisfactory solution for the same reason that encoded names are not quite a satisfactory solution to the funarg problem; they are clumsy, and the possibility still remains of tripping over one of them. What is desired is for a program to be able to interact with the outside world in only one way: through the top level

functions.

One somewhat less than satisfactory solution would be to borrow the idea of a "program module", which hides all but the requested function names. Again assuming deep binding and the Section 9 alist structure, we could define the MODULE function with the following syntax:

```
(MODULE <list of environments>
        (<function name 1> <function 1>)
        (<function name 2> <function 2>)
        ...
        (<function name n> <function n>))
```

where the <list of environments> and <function i> parameters are evaluated. This creates and returns an alist segment which, for $1 \leq k \leq n$, binds <function name k> to a version of <function k> which has a closure consisting of copies of the environments in <list of environments> and, circularly, the bindings of <function name i> for $1 \leq i \leq n$.

For example, the call:

```
(MODULE (LIST '((A B) . (1 2)) '((C D) . (3 4)))
        (FN1 (LAMBDA (X) (FN2 (* C (SUB1 X)))))
        (FN2 (LAMBDA (Y) (PRINT (+ B Y))))
        (FN3 (FLAMBDA (Z W) (CONS W Z))))
```

would return the hopelessly involved structure depicted in Figure 11.

This structure has the property that the variables A, B, C, D, FN1, FN2, and FN3 are part of the module in that the effects of any SETQ's on them will remain in the structure. This is certainly reasonable; the same thing will happen in a

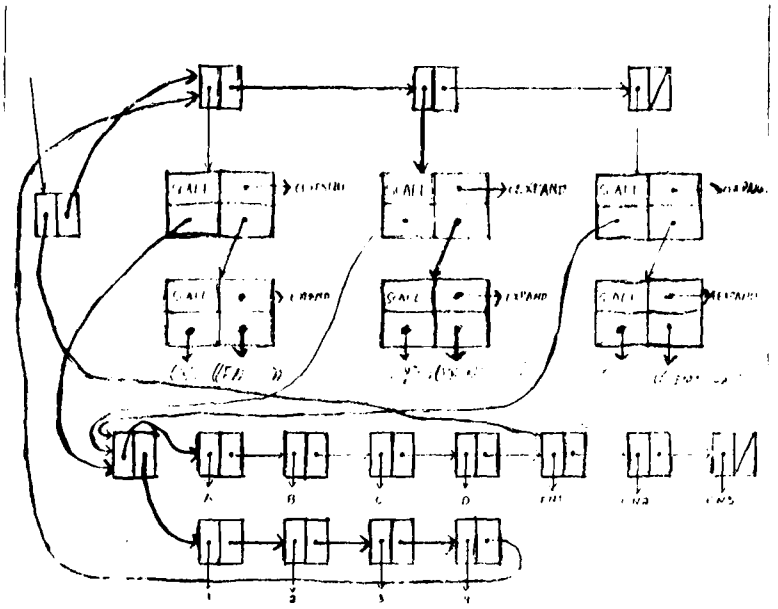


Figure 11

FUNCTION-trapped environment. In the latter case, this effect can serve as a means of communication between functions in different environments.

To use this facility, one breaks up the set of functions being defined into convenient, conceptually related subsets, each of which should be made into a module. As an example, a program might be organized this way:

```
(SETQ PRIMITIVES (MODULE NIL
        (<primitive 1> <defn of primitive 1>)
        ...
        (<primitive n> <defn of primitive n>)))
(SETQ LOW-LEVEL-PACKAGE-1 (MODULE (LIST PRIMITIVES)
        (<low level function 1> <its defn>)
        ...
        (<low level function n> <its defn>)))
...
(SETQ LOW-LEVEL-PACKAGE-19 (MODULE (LIST PRIMITIVES)
        ...))
(SETQ FROTZ-PACKAGE (MODULE
        (LIST LOW-LEVEL-PACKAGE-1 ...
```

```
        LOW-LEVEL-PACKAGE-19)
    (TOP-LEVEL-FUNCTION-1 <its defn>)
    ...
    (TOP-LEVEL-FUNCTION-5 <its defn>)))
```

While this bizarre scheme solves the problems of function reference discussed above, it is rather intimately tied to the assumption of deep binding. It therefore carries with it the major problem with deep binding: it is prone to large value lookup times. Since function names are almost always free variables, the search time for them will be the greatest. This is an important consideration when one is deciding between deep and shallow binding in a LISP system. Shallow binding allows fast and easy lookup of values, while deep binding allows fast and easy changes of environments.

9. How to Build an Association List

Besides the means of creating and manipulating functional objects, another topic in LISP implementations which varies between various LISP implementations is the question of how one denotes and manipulates functions which can take varying numbers of arguments. A general and convenient syntax and implementation is described here.

Recall the assumption about the structure of the alist which was made when introducing LAMBDA, EXPR, and so forth in Chapter 4. The alist was assumed to be a list of alist "segments", each of the form (<variables> . <their values>). The code for EXPAND took the argument list directly as specified in the definition of the function and made it into the <variables> entry for the new segment of the alist. If we adopt a convenient notation for optional and extra arguments, we can keep this alist structure by just implementing a slightly more complex alist lookup mechanism.

The suggested syntax for LAMBDA is as follows:

A LAMBDA-expression has the form (LAMBDA <arglist> <body1> ... <bodyn>), where
<arglist> -> <variable> | <varlist> | <extended varlist>
<varlist> -> (<variable>*)
<extended varlist> -> (<variable>* . <variable>)
<variable> -> <atomic symbol>

For example, if we do (SETQ FOO (LAMBDA (A B . C) (LIST A B C))), then (FOO 'X 'Y 'Z 'W) will return (X Y (Z W)), because, inside FOO, A will be bound to X, B will be bound to Y, and C will be

bound to a list of the remaining arguments, (Z W). We could also define the LIST function by (SETQ LIST (LAMBDA ARGS ARGS)), because ARGS would be bound to a list of all the arguments given to LIST (once they were evaluated).

To implement this, the code which binds arguments to their values on the alist looks like:

```
alist <- cons(cons(<arg list>,<arg values>),alist)
```

The code for the alist search algorithm is:

```
lookup(var)
  seq <- alist
  WHILE seq ≠ nil DO
    varlst <- car(car(seq))
    vallst <- cdr(car(seq))
    WHILE varlst is not an atom AND vallst ≠ nil DO
      IF car(vallst) = var
        THEN return(car(vallst))
        ELSE varlst <- cdr(varlst)
             vallst <- cdr(vallst)
      ENDIF
    ENDWHILE
    IF varlst = var
      THEN return(vallst)
    ENDIF
    seq <- cdr(seq)
  ENDWHILE
  yell()   /* variable is not on alist */
```

This algorithm has some interesting consequences. For example, if some random piece of code needs to place a single variable-value pair on the alist, it need only do:

```
alist <- cons(cons(variable,value),alist)
```

Also, the alist is built partially of cons nodes which are part of function definitions, since the argument lists are not

analyzed in any way; they are simply consed into the alist. In the case of special forms and macros, the value list is also made part of the alist, but this might not be entirely desirable since many SETQ mechanisms RPLACA alists (and thus the source code for the special form calls). Thus this scheme saves cons nodes in many situations without excessively slowing the lookup mechanism. (This is not quite true if the implementation in question requires more than a couple of instructions to determine whether a given pointer's referent is an atom.)

10. Conclusion

This paper has presented a theory and implementation of functions which are to be considered as data objects. Some fairly important points have been made which deserve to be summarized:

1. It is both reasonable and desirable to treat functions as data objects. This is approached from the view of abstract data structures.

2. The major existing implementations of LISP do not make the distinction between internal and external representations of functions which is necessary to make comprehensible the concept of functions as data objects.

3. Programmers wish to deal with several types of functions. Implementation can be eased by analyzing the nature of the differences between these function types. In this somewhat idealized exposition, the differences were decomposed into: (i) the manner of evaluation of arguments, (ii) the nature and content of the environment in which the body is to be evaluated, and (iii) the manner of evaluation of the body. This decomposition is reflected in the way in which the interpreter's various internal variables were handled.

4. Function-based computation, if effectively implemented, could be a valuable programming tool. Expressing complex

operations as operators on functions can effectively hide control constructs which are not necessary for exposition.

The discussion of Section 8 has made it clear that the last word has not yet been written on the subject. What has been shown is that it is possible to improve the theoretical basis of current implementations of LISP, and in doing so ease the task of the programmer who must deal with complicated algorithms and control structures.

11. Bibliography

[Allen] Allen, John, "Anatomy of LISP", McGraw-Hill, New York, 1978

[Church] Church, Alonzo, "The Calculi of Lambda-conversion", Annals of Mathematics Studies, Princeton University Press, 1941; Reprinted by Klaus Reprint Corp., New York, 1965

[McCarthy1] McCarthy, John, et al., "LISP 1.5 Programmer's Manual", Second Edition, MIT Press, Cambridge, 1965

[McCarthy2] McCarthy, John, "History of LISP", in "Proceedings of the SIGPLAN History of Programming Languages Conference", June 1978

[Norman] Norman, Eric, "Documentation for 1100 LISP Implementation", unpublished paper, University of Wisconsin at Madison, 1976

[Steele] Steele, Guy, "Rabbit: A Compiler for Scheme", MIT AI Lab Report AI-TR-474, May 1978

[Steele&Sussman] Steele, Guy, and Gerald Jay Sussman, "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two)", MIT AI Lab Memo 453, May 1978

12. Appendix - A LISP Rendering of the Scheme

The LISP code presented here simulates the typing functions and evaluator presented in this paper. It is written in Wisconsin LISP, which already possesses most of the characteristics this system is supposed to implement, but the code does provide a further understanding of the intent of the algorithms.

```
?
? The One True LISP Interpreter
? Phil Agre, January 1979
?
? The data type codes used are: 0 is cons node, 1, 2, and 3 are
? numbers, 4 is system code, 5 is compiled code, 6
? is linker node, 7 is atomic symbol, and 8 is string.
?
? The typing functions will take either lTLI-defined functions
? or Wisconsin LISP-defined functions, but not DEFSPEC's,
? DEFMAC's, or traced versions of either of these.
? Other bugs probably abound.
?
? A crude structure scanning readmacro package is used here.
? /<sexpr> expands into a call on S-STRUCTSCAN which evaluates
? all expressions which have S-SCANFLAG's consed onto them,
? where ,<sexpr> expands into (S-SCANFLAG . <sexpr>).
?
? The lTLI versions of LAMBDA and FUNCTION are called XLAMBDA
? XFUNCTION, resp., to avoid collisions with the Wisconsin
? LISP names.  Also, PROG's are not handled by lTLI.
?
? The lTLI evaluator is called XEVAL (so as not to conflict with
? the existing EVAL function).
?
(CSETQ XEVAL (LAMBDA (X-SEXPR)
     (COND ((ATOM X-SEXPR) (EVAL X-SEXPR))
           ((AND (EQUAL (TYPE (CAR X-SEXPR)) 7)
                 (CSETQ ::T1 (GET (CAR X-SEXPR) 'FEXPR)))
              (::T1 (STACK (CDR X-SEXPR))))
           ((EQ (CAR X-SEXPR) 'QUOTE) (CADR X-SEXPR))
           ((EQ (CAR X-SEXPR) 'COND)
              (XCOND (CDR X-SEXPR)))
           ((EQ (CAR X-SEXPR) 'CSETQ)
              (CSET (CADR X-SEXPR) (XEVAL (CADDR X-SEXPR))))
           ((EQ (CAR X-SEXPR) 'SETQ)
              (SET (CADR X-SEXPR) (XEVAL (CADDR X-SEXPR))))
           ((MEMBER (CAR X-SEXPR) FEXPR-LIST)
```

```
            (EVAL X-SEXPR))
          ((AND (ATOM (CAR X-SEXPR))
                (MEMBER (TYPE (EVAL (CAR X-SEXPR))) '(4 6)))
           ((EVAL (CAR X-SEXPR))
            (STACK (INTO (CDR X-SEXPR) XEVAL))))
          (T (XEVAL1 (XEVAL (CAR X-SEXPR)) (CDR X-SEXPR)
                0 0 0 0 0)) )))
?
? XEVAL1 establishes the local variables for the other
? routines to manipulate.
?
(CSETQ XEVAL1 (LAMBDA (X-FN X-ARGS X-EVARGS X-LABPAIR
                       X-ALTENV X-CLOSE X-CONT)
    (XEVAL X-FN)))
?
? (APPLY <fn> <arglist>) executes the function <fn> with
? the given arguments without evaluating them.
? (STACK is a thoroughly disgraceful hack which
? takes as its argument a list of objects and shuffles
? the stack so that the surrounding function will appear
? to have been given those objects as arguments.)
?
(CSETQ APPLY (LAMBDA (X-FN X-ARGS)
    (COND ((MEMBER (TYPE X-FN) '(4 6))
           (X-FN (STACK X-ARGS)))
          (T (XEVAL1 X-FN 0 X-ARGS 0 0 0 0))) ))
?
? A list of the special forms which must be handled
? specially by the interpreter.  (These are functions
? defined by Wisconsin LISP, not Fexpr's defined through
? the lTLI interpreter.)
?
(CSETQ FEXPR-LIST '(ZEROSET LAMBDA LAMDA DEFSPEC DEFMAC
    S-STRUCTSCAN))
?
? This executes COND's for lTLI.
?
(CSETQ XCOND (LAMBDA (X-CONDS)
    (PROG (VAL)
          (MAPC X-CONDS (LAMBDA (X-CLAUSE)
              (SETQ VAL (XEVAL (CAR X-CLAUSE)))
              (COND (VAL (RETURN (XDO (CDR X-CLAUSE))))))))
          (RETURN NIL))))
?
? Provisionally define the LAMBDA's.
?
(MAPC '((XLAMBDA . EXPAND) (FLAMBDA . FEXPAND1)
                           (MLAMBDA . MEXPAND1))
      (LAMBDA (X-PR)
          (EVAL /(DEFSPEC ,(CAR X-PR) (LAMBDA (X-ARGS . L-EXPRS)
                    (LIST ',(CDR X-PR) X-ARGS L-EXPRS)))) ))
?
? To define FEXPR's which must be interpretable by lTLI but
? defined in Wisconsin LISP, we revert to the hack of putting
```

```
? function definitions on property lists.  It should be
? emphasized that this has nothing to do with the real
? implementations of the lTLI functions.
?
(DEFSPEC PSEUDOFEXPR (LAMBDA (X-NAME X-FN)
     (PUT X-NAME 'FEXPR (EVAL X-FN))))
?
? Define (LABEL <atom> <fn>) to return a version of <fn>
? which will always be run in an environment in which
? <atom> is bound to <fn>.
?
(PSEUDOFEXPR LABEL (LAMBDA (X-ATM X-FN)
     /(LEXPAND ,X-ATM ,(XEVAL X-FN))))
?
? (XFUNCTION <fn>) is MacLISP's (*FUNCTION <fn>).  It
? grabs the current alist and makes it part of the
? version of <fn> which is returned.
?
(CSETQ XFUNCTION (XLAMBDA (X-FN)
     /(ENVEXPAND ,(CLEANALIST (ALIST)) ,X-FN)))
?
? (CLOSURE '(<atm1> ... <atmn>) <fn>) constructs associations
? between the <atmi>'s and their current values and attaches
? them to the version of <fn> which is returned.  All
? other functions are allowed to retain the dynamic
? binding mechanism.
?
(CSETQ CLOSURE (XLAMBDA (X-VARS X-FN)
     /(CEXPAND ,(INTO X-VARS (LAMBDA (X-VAR)
                               (ASSOC X-VAR (ALIST))))
               ,X-FN)))
?
? This is the expansion routine for EXPR.
?
(PSEUDOFEXPR EEXPAND (LAMBDA (X-FN)
     (ZEROSET X-CONT XDO)
     (COND ((ZEROP X-ARGS))
           (T (SETQ X-EVARGS (INTO X-ARGS XEVAL))
              (SETQ X-ARGS 0)))
     (EXECUTE X-FN)))
?
? This is the expansion routine for FEXPR.
?
(PSEUDOFEXPR FEXPAND (LAMBDA (X-FN)
     (ZEROSET X-CONT XDO)
     (MOVEARGS)
     (EXECUTE X-FN)))
?
? This is the expansion routine for MACRO.
?
(PSEUDOFEXPR MEXPAND (LAMBDA (X-FN)
     (ZEROSET X-CONT (LAMBDA (X-SEXPRS) (XEVAL (XDO X-SEXPRS))))
     (MOVEARGS)
     (EXECUTE X-FN)))
```

```
?
? This is the expansion routine for LAMBDA.
?
(PSEUDOFEXPR EXPAND (LAMBDA (X-ARGNAMES X-EXPRS)
     (ZEROSET X-CONT XDO)
     (EVALEXPRS X-EXPRS (APPEND (MATCHUP X-ARGNAMES (XARGLIST))
                                (XALIST (CDDR (ALIST))) ))))
?
? This is the expansion routine for FLAMBDA.
?
(PSEUDOFEXPR FEXPAND1 (LAMBDA (X-ARGNAMES X-EXPRS)
     (MOVEARGS)
     (ZEROSET X-CONT XDO)
     (EVALEXPRS X-EXPRS (APPEND (MATCHUP X-ARGNAMES X-EVARGS)
                                (XALIST (CDDR (ALIST)))))))
?
? This is the expansion routine for MLAMBDA.
?
(PSEUDOFEXPR MEXPAND1 (LAMBDA (X-ARGNAMES X-EXPRS)
     (MOVEARGS)
     (ZEROSET X-CONT (LAMBDA (X-SEXPRS) (XEVAL (XDO X-SEXPRS))))
     (EVALEXPRS X-EXPRS (APPEND (MATCHUP X-ARGNAMES X-EVARGS)
                    (XALIST (CDDR (ALIST)))))))
?
? This is the expansion routine for LABEL.
?
(PSEUDOFEXPR LEXPAND (LAMBDA (X-ATM X-FN)
     (SETQ X-LABPAIR (CONS X-ATM X-FN))
     (EXECUTE X-FN)))
?
? This is the expansion routine for CLOSURE.
?
(PSEUDOFEXPR CEXPAND (LAMBDA (X-VARS X-FN)
     (COND ((ZEROP X-CLOSE) (SETQ X-CLOSE X-VARS))
           (T (SETQ X-CLOSE (APPEND X-VARS X-CLOSE))))
     (EXECUTE X-FN)))
?
? This is the expansion routine for XFUNCTION.
?
(PSEUDOFEXPR ENVEXPAND (LAMBDA (X-ALST X-FN)
     (ZEROSET X-ALTENV X-ALST)
     (EXECUTE X-FN)))
?
? The EXPR, FEXPR, and MACRO functions each take a function
? as an argument and return a version of that function
? which is an expr, a fexpr, or a macro, resp.
?
(MAPC '((EXPR . EEXPAND) (FEXPR . FEXPAND) (MACRO . MEXPAND))
      (LAMBDA (X-PR)
          (EVAL /(CSETQ ,(CAR X-PR) (XLAMBDA (X-FN)
                     (LIST ',(CDR X-PR) X-FN)))) ))
?
? Redefine the LAMBDA's so that they can be interpreted by lTLI.
? (Re-define FLAMBDA last.)
```

```
?
(MAPC '((XLAMBDA . EXPAND) (MLAMBDA . MEXPAND1)
                                    (FLAMBDA . FEXPAND1))
      (LAMBDA (X-PR)
          (EVAL /(CSETQ ,(CAR X-PR) (FLAMBDA (X-ARGS . L1-EXPRS)
                    (LIST ',(CDR X-PR) X-ARGS L1-EXPRS)))) ))
?
? If the arguments have not already been processed,
? then (MOVEARGS) will cause them to be moved to
? X-EVARGS unevaluated.
?
(CSETQ MOVEARGS (LAMBDA NIL
     (COND ((ZEROP X-ARGS))
           (T (SETQ X-EVARGS X-ARGS)
              (SETQ X-ARGS 0)))))
?
? (ZEROSET <var> <val>) is "if <var>=0 then <var> <- <val>".
?
(DEFSPEC ZEROSET (LAMBDA (X-VAR X-VAL)
     (COND ((ZEROP (EVAL X-VAR)) (SET X-VAR (EVAL X-VAL))) )))
?
? EXECUTE could be replaced by XEVAL except for the
? possibility of having non-lTLI functions floating
? around.  We note that XEVAL'ing a function here is
? the analog of jumping to it in the assembly language
? code.
?
(CSETQ EXECUTE (LAMBDA (X-FN)
     (COND ((MEMBER (TYPE X-FN) '(4 6))
              (X-FN (STACK (XARGLIST))))
           (T (XEVAL X-FN)))))
?
? XARGLIST evaluates the arguments if they have not
? yet been processed.
?
(CSETQ XARGLIST (LAMBDA NIL
     (COND ((ZEROP X-ARGS) X-EVARGS)
           (T (INTO X-ARGS XEVAL)) )))
?
? This function binds variables to their values.
? See the section on alist implementation.
?
(CSETQ MATCHUP (LAMBDA (X-NAMES X-VALS)
     (COND ((NULL X-NAMES) NIL)
           ((ATOM X-NAMES) (LIST (CONS X-NAMES X-VALS)))
           ((ATOM X-VALS) (PRINT "NOT ENOUGH ARGUMENTS") NIL)
           (T (CONS (CONS (CAR X-NAMES) (CAR X-VALS))
                    (MATCHUP (CDR X-NAMES) (CDR X-VALS)))) )))
?
? Given the body of a function and an alist, this
? function does the evaluation in the presence of the
? given alist.
?
(CSETQ EVALEXPRS (LAMBDA (X-EXPRS X-ALST)
```

```
     (EVAL /(,X-CONT ',X-EXPRS) X-ALST)))
?
? (XDO <list-of-s-expressions>) evaluates each s-expression
? and returns the value of the last one.
?
(CSETQ XDO (LAMBDA (X-EXPRS)
     (COND ((ATOM X-EXPRS) NIL)
           ((ATOM (CDR X-EXPRS)) (XEVAL (CAR X-EXPRS)))
           (T (XEVAL (CAR X-EXPRS)) (XDO (CDR X-EXPRS))) )))
?
? XALIST takes a sanitized version of the current alist and
? figures out what alist to use in evaluating the body
? of the current function.
?
(CSETQ XALIST (LAMBDA (X-ALST)
     (COND ((ZEROP X-ALTENV))
           (T (SETQ X-ALST X-ALTENV)))
     (COND ((ZEROP X-LABPAIR))
           (T (SETQ X-ALST (CONS X-LABPAIR X-ALST))))
     (COND ((ZEROP X-CLOSE))
           (T (SETQ X-ALST (APPEND X-CLOSE X-ALST))))
     (CLEANALIST X-ALST)))
?
? Mark all the variables which should not be involved in
? trapped environments.
?
(CSETQ INTERNAL-VARS '(X-SEXPR X-FN X-ARGS X-EVARGS X-LABPAIR
     X-ALTENV X-CLOSE X-CONT X-CONDS X-VAL X-ATM X-VARS
     X-ARGNAMES X-EXPRS X-ALST X-PR X-VAR X-CLAUSE X-NAME
     S-STRUCT))
(MAPC INTERNAL-VARS (LAMBDA (VAR) (FLAG VAR 'INTERNAL-VAR)))
?
? CLEANALIST takes an association list and removes
? from it the bindings of internal variables.
?
(CSETQ CLEANALIST (LAMBDA (X-STARTALIST)
     (PROG ((X-ALST X-STARTALIST) (X-RES '(CLEAN)))
      LOOP (COND ((NULL X-ALST)
                  (RETURN (REVERSE X-RES)))
                 ((EQ (CAR X-ALST) 'CLEAN)
                  (RETURN (APPEND (REVERSE X-RES)
                                  (CDR X-ALST))))
                 ((ATOM (CAR X-ALST)) (GO DECR))
                 ((IFFLAG (CAAR X-ALST) 'INTERNAL-VAR)
                  (GO DECR)))
           (SETQ X-RES (CONS (CAR X-ALST) X-RES))
       DECR (SETQ X-ALST (CDR X-ALST))
           (GO LOOP))))
?
? This is the interpreter.
?
(CSETQ XINTERPRETER (LAMBDA NIL
     (PRINT " ")
     (PRINT "ONE TRUE LISP INTERPRETER 1.0")
```

```
(PRINT " ")
((LAMBDA (X-CONT)
    (LISP (LAMBDA NIL
              (SETQ X-CONT 0)
              /',(XEVAL (*READ "XEVAL:")))))
 0)))
```